

# MEMORY-ADJUSTABLE NAVIGATION PILES WITH APPLICATIONS TO SORTING AND CONVEX HULLS\*

OMAR DARWISH<sup>†</sup>, AMR ELMASRY<sup>‡</sup>, AND JYRKI KATAJAINEN<sup>§</sup>

**Abstract.** We consider space-bounded computations on a random-access machine (RAM) where the input is given on a read-only random-access medium, the output is to be produced to a write-only sequential-access medium, and the available workspace allows random reads and writes but is of limited capacity. The length of the input is  $N$  elements, the length of the output is limited by the computation, and the capacity of the workspace is  $O(S)$  bits for some predetermined parameter  $S$ . We present a state-of-the-art priority queue—called an adjustable navigation pile—for this restricted RAM model. Under some reasonable assumptions, our priority queue supports *minimum* and *insert* in  $O(1)$  worst-case time and *extract* in  $O(N/S + \lg S)$  worst-case time for any  $S \geq \lg N$ . (We use  $\lg x$  as a shorthand for  $\log_2(\max\{2, x\})$ .) We show how to use this data structure to sort  $N$  elements and to compute the convex hull of  $N$  points in the two-dimensional Euclidean space in  $O(N^2/S + N \lg S)$  worst-case time for any  $S \geq \lg N$ . Following a known lower bound for the space-time product of any branching program for finding unique elements, both our sorting and convex-hull algorithms are optimal. The adjustable navigation pile has turned out to be useful when designing other space-efficient algorithms, and we expect that it will find its way to yet other applications.

## 1. Introduction.

**1.1. Problem Area.** Consider a sequential-access machine (Turing machine) that has three tapes: input tape, output tape, and work tape. In space-bounded computations, the input tape is read-only, the output tape is write-only, and the aim is to limit the amount of space used in the work tape. In this set-up, the theory of language recognition and function computation requiring  $O(\lg N)$  bits of working space for an input of size  $N$  is well established; people talk about log-space programs [40, Section 3.9.3] and classes of problems that can be solved in log-space [40, Section 8.5.3]. Also, in this set-up, trade-offs between space and time have been extensively studied [40, Chapter 10]. Although one would seldom be forced to rely on a log-space program, it is still theoretically interesting to know what can be accomplished when only a logarithmic number of extra bits are available.

In this paper we reconsider the space-time trade-offs on a random-access machine (RAM). Analogous with the sequential-access machine, we have separate storage media for read-only input, write-only output, and read-write workspace which is of limited capacity. Now, however, both the input and workspace allow random access, but the output is still to be produced sequentially. Over the years, starting by a seminal paper of Munro and Paterson [33]—where a related model was used, the space-time trade-offs in this *restricted RAM model* have been studied for many problems including: sorting [21, 37], selection [19, 21], and various geometric problems [2, 4, 12, 13]. The practical motivation for some of the previous work has been the appearance of special devices, where the size of working space is limited (e.g. mobile devices) and where writing is expensive (e.g. flash memories).

An algorithm or a data structure is said to be *memory adjustable* if it uses  $O(S)$  bits of working space for a given parameter  $S$ . Naturally, we expect to use at least a constant number of words, so  $\Omega(w)$  is a lower bound for the space usage,  $w$  being the size of the machine word in bits. Sorting is one of the few problems for which

\*Parts of this paper have appeared in preliminary form in [3] and [11]

<sup>†</sup>Max-Planck Institute for Informatics, Saarbrücken, Germany

<sup>‡</sup>Department of Computer Engineering and Systems, Alexandria University, Egypt

<sup>§</sup>Department of Computer Science, University of Copenhagen, Denmark

the optimal space-time product has been settled: Beame showed [5] (see also [40, Theorem 10.13.8]) that  $\Omega(N^2)$  is a lower bound, and Pagter and Rauhe showed [37] that an  $O(N^2/S + N \lg S)$  worst-case running time is achievable for any  $S \geq \lg N$ . As for the convex-hull problem, Chan and Chen [9] gave an algorithm for computing the convex hull of a planar set of  $N$  points, stored in a read-only array, that runs in  $O((N^2/S) \cdot \lg N + N \lg S)$  worst-case time for any  $S \geq \lg N$ .

**1.2. Model of Computation.** We assume that the elements being manipulated are on a read-only array, and use  $N$  to denote the number of elements stored there. Observe that  $N$  does *not* need to be known beforehand. The output is sent to a separate write-only stream, where the output printed cannot be read or rewritten.

In addition to the input and output media, a limited random-access workspace is available. The data on this workspace is manipulated wordwise as on the word RAM [23]. We assume that the word size  $w$  is at least  $\lceil \lg N \rceil$  bits and that the processor is able to execute the same arithmetic, logical, and bitwise operations as those supported by contemporary imperative programming languages—like C [29]. It is a routine matter [31, Section 7.1.3] to store a bit vector of size  $N$  such that it occupies  $\lceil N/w \rceil$  words and any string of at most  $w$  bits can be accessed in  $O(1)$  worst-case time. That is, the *time complexity* of an algorithm is proportional to the number of primitive operations plus the number of element accesses and element comparisons performed. We do *not* assume the availability of any powerful memory-allocation routines. The workspace is an infinite array (of words), and the space used by an algorithm is the prefix of this array. Even though this prefix can have some unused zones, the length of the whole prefix specifies the *space complexity* of the algorithm.

In our setting, the elements lie in a read-only array and the data structure only constitutes references to these elements. We assume that each of the elements appears in the data structure at most once, and it is the user’s responsibility to make sure that this is the case. Also, all operations are position-based; the position of an element can be specified by its index. Since the positions can be used to distinguish the elements, we implicitly assume that the elements are distinct.

**1.3. Our Results.** Let  $A$  be a read-only array and let  $N$  denote its size. Consider a priority queue  $Q$  storing a subset of the elements in  $A$ . We use  $|Q|$  to denote the number of elements in  $Q$ . In the adjustable set-up, a (*min*-)priority queue is a data structure that supports the following operations:

$Q.minimum()$ : Return the index of the minimum element in  $Q$ , as long as  $|Q| > 0$ .

$Q.insert(i)$ : Insert  $A[i]$  into  $Q$ , for some  $i \in \{0, 1, \dots, N - 1\}$ .

$Q.extract(i)$ : Extract  $A[i]$  from  $Q$ , for some  $i \in \{0, 1, \dots, N - 1\}$ .

A *max-priority queue*, which is defined to support the operation *maximum* instead of *minimum*, is obtained from a min-priority queue by reversing the comparison function used in element comparisons. In the non-adjustable set-up, any priority queue—like a binary heap [41] or a queue of pennants [7] (that both operate in-place)—could be used to store positions of the elements instead of the elements themselves.

In the first part of the paper, we improve and simplify the memory-adjustable priority queue presented by Pagter and Rauhe [37] by introducing a kindred data structure that we call an *adjustable navigation pile*. Compared to the navigation piles of [28], that require  $\Theta(N)$  bits, our adjustable variant can achieve the same asymptotic run-time performance with only  $\Theta(N/\lg N)$  bits. (Another priority queue that uses  $\Theta(N)$  bits in addition to the input was given in [16].) In Table 1, we compare the performance of the new data structure to some of its competitors. Note that the stated bounds are valid under some reasonable assumptions declared in Section 2.

TABLE 1

The performance of adjustable navigation piles and their competitors in the restricted RAM model;  $N$  is the size of the read-only input and  $S$  is an asymptotic target for the size of workspace in bits where  $S \geq \lg N$ .

Reference	Space	minimum	insert	extract
[7]	$\Theta(N \lg N)$	$O(1)$	$O(1)$	$O(\lg N)$
[28]	$\Theta(N)$	$O(1)$	$O(\lg N)$	$O(\lg N)$
[21]	$\Theta(S)$	$O(1)$	$O((N \lg N)/S + \lg S)$	$O((N \lg N)/S + \lg S)$
[37]	$\Theta(S)$	$O(N/S^2 + \lg S)$	$O(N/S + \lg S)$ amortized	$O(N/S + \lg^2 S)$
[this paper]	$\Theta(S)$	$O(1)$	$O(1)$	$O(N/S + \lg S)$

In the second part of the paper, we use the adjustable navigation pile for sorting. Our algorithm is simpler and more intuitive than that of Pagter and Rauhe [37], and we also achieve the optimal  $O(N^2/S + N \lg S)$  running time for any  $S \geq \lg N$ . The algorithm is priority-queue sort like heapsort [41]: Insert the  $N$  elements one by one into a priority queue and extract the minimum from that priority queue  $N$  times.

In the third part of the paper, we improve the Chan-Chen bound for the convex-hull problem by introducing an algorithm that runs in  $O(N^2/S + N \lg S)$  time for any  $S \geq \lg N$ . To prove this result, we augment the adjustable navigation pile with extra information while still using  $O(S)$  bits of workspace.

**1.4. Related Models.** The basic feature that distinguishes the model we use from other related models is the capability of having random access to the input data. In the context of sequential-access machines, the input is on a tape that only allows single-pass algorithms. The so-called *streaming model* still enforces sequential access, but allows multi-pass algorithms, and the goal is to minimize the number of passes over the input when the size of the random-access workspace is limited. Munro and Paterson [33] considered this model. For some problems, the restricted RAM model is more powerful than the multi-pass streaming model. For example, for the selection problem the lower bound known for the multi-pass streaming model [8] can be bypassed in the restricted RAM model [19].

In the *in-place model*, the elements are to be stored at the beginning of an infinite array, a constant number of additional variables are allowed, and the elements may be swapped and overwritten, but not modified, still keeping this compact representation. All the problems considered in this paper—maintaining priority queues (see, e.g. [7, 15, 41]), sorting (see, e.g. [15, 27, 41]), and computing convex hulls [6]—have been studied in this classical setting. In the *restore model* [10], the input elements may be temporarily rearranged, and even modified, during a computation, but at the end the input must be restored to its original state. Again, sorting problems have been central in the exploration of the power of this model (see [10, 26]). In general, any reversible algorithm would work well in the restore model, so reversible computing is distantly related to this study.

**1.5. Bit Vectors with rank and select Support.** Given a bit vector  $B$  of  $N$  bits, of which  $n$  are 1 bits, consider the following operations:

$B.access(i)$ : Return  $B[i]$ , i.e. the bit at index  $i$  for some  $i \in \{0, 1, \dots, N-1\}$ .

$B.rank(i)$ : Return the number of 1 bits among the bits  $B[0], B[1], \dots, B[i]$ , for some  $i \in \{0, 1, \dots, N-1\}$ .

$B.select(j)$ : Return the index of the  $j$ th 1 bit, for some  $j \in \{1, 2, \dots, n\}$ , i.e. if  $B.select(j) = i$ , this means that  $B[i] = 1$  and  $B.rank(i) = j$ .

The operations  $B.rank0(i)$  and  $B.select0(j)$  are similarly defined considering the 0 bits instead of the 1 bits.

For a bit vector of size  $N$ , our requirements for an acceptable solution are that all the operations should run in  $O(1)$  worst-case time, the space used should be  $O(N)$  bits, and the construction of the data structure should take  $O(N)$  worst-case time. The problem of extending a bit vector with *rank-select* operations has been addressed in several papers (see, for example, [24, 32, 39]). Most of the known solutions rely on the idea of dividing the bit vector into blocks, precomputing *rank* and *select* values for some specific positions, and calculating the other values on the fly using the stored values, some precomputed tables, and bits in the bit vector itself. For example, the solution presented in [39] would be suitable for our purposes; it requires  $N + O(N \lg \lg N / \lg N)$  bits, and  $O(1)$  worst-case time per operation.

## 2. Memory-Adjustable Priority Queues.

**2.1. Assumptions.** In this section two memory-adjustable priority queues are described. The first structure is a straightforward adaptation of a tournament tree (also called a selection tree [30, Section 5.4.1]) for read-only data. For a parameter  $S$ , it uses  $O(S)$  words of workspace. The second structure is an improvement of a navigation pile [28] for which the workspace is  $O(S)$  bits, for  $S \geq \lg N$ , where  $N$  is the size of the read-only input. Both data structures can perform *minimum* and *insert* in  $O(1)$  worst-case time and *extract* in  $O(N/S + \lg S)$  worst-case time.

When describing the data structures, we make the following assumptions:

1.  $N$  is known beforehand.
2. Insertions and extractions are *monotonic* such that, at any given point of time, there is a single element (if any) indicating that the elements smaller than or equal to it are outside the data structure. We call such an element the *latest output*, and say that an element is *alive* if it is larger than the latest output. In particular, an extraction must remove the smallest element and an insertion must add an element that is larger than the latest output.
3. Insertions are *sequential*—but insertions and extractions can be intermixed—such that the elements are inserted one by one from consecutive input entries starting from the first element stored in the read-only input.

These assumptions are valid when a priority queue is used for sorting. Actually, in sorting all insertions are executed before extractions; a restriction that is not mandated by the data structure. At the end of this section, we show how to get rid of these assumptions. The first assumption is not critical. But, when relaxing the second assumption, the required size of workspace has to increase by  $N$  bits. When relaxing the third assumption, the asymptotic worst-case running time of *insert* will become the same as that required by *extract*.

**2.2. Adjustable Tournament Trees.** For an integer  $S$ , we use  $\bar{S}$  as a shorthand for  $2^{\lceil \lg S \rceil}$ . It suffices that  $S \leq N / \lg N$ ; even if  $S$  was larger, the operations would not be asymptotically faster. The input array is divided into  $\bar{S}$  *buckets* each containing  $\lceil N / \bar{S} \rceil$  elements, except for the last bucket that may contain less. A complete binary tree is built above these buckets. Each leaf of this tree *covers* a single bucket and each branch covers the buckets of the leaves in the subtree rooted at that branch. We call the elements within the buckets covered by a node the *covered range* of this node. Note that the covered range of a node is a sequence of elements stored in consecutive locations of the input array. The data stored at each node is an index specifying the position of the smallest alive element in the covered range of that node.

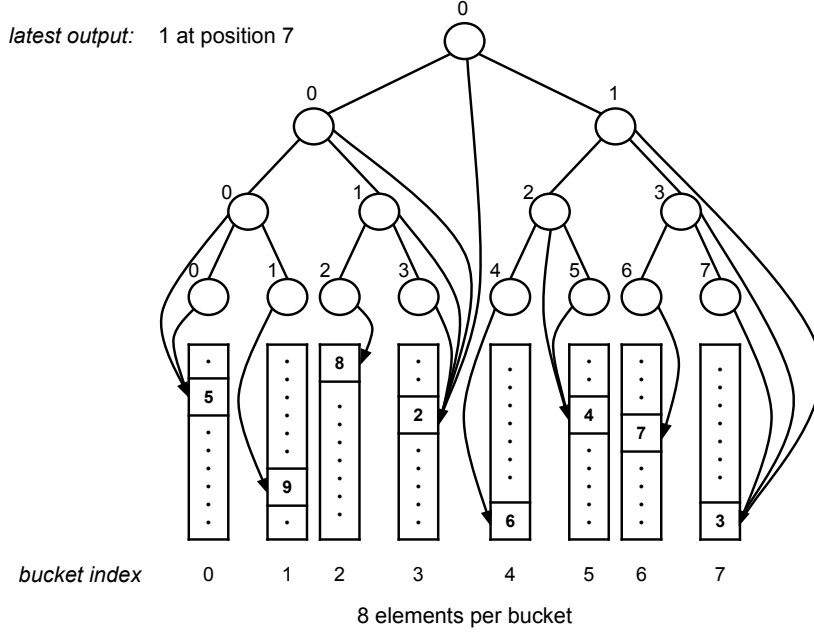


FIG. 1. An adjustable tournament tree for a set of  $N = 64$  elements when  $\bar{S} = 8$ . Indices stored at the nodes are visualized as pointers. Only the smallest alive element in each bucket is shown.

An *adjustable tournament tree* is an array of  $2\bar{S} - 1$  positions (indices). To make the connection to our adjustable navigation piles clear, we store the indices in this array in breadth-first order as in a binary heap [41]. At each level, we start the indexing of the nodes from 0. For the sake of simplicity, we maintain a *header* that stores the offsets to the beginning of each level (even though this information is easy to calculate). For a node number  $i$ , its left child has number  $2i$  at the level below, the right child has number  $2i + 1$  at the level below, and the parent has number  $\lfloor i/2 \rfloor$  at the level above. When we know the current level and the index of a node at that level, the information available at the header and the above formulas are enough to get to a neighbouring node in constant time. In Figure 1, we give an illustration of an adjustable tournament tree for a set of  $N = 64$  elements when  $\bar{S} = 8$ .

To support *insert* efficiently, we partition the data structure into three components: tournament tree, submersion buffer, and insertion buffer. The *submersion buffer* is the last full bucket that is being submerged into the tournament tree. The *insertion buffer* is the bucket that embraces the new elements. One or both of these buffers can be empty. The idea is to insert the elements into a buffer and, when the buffer gets full, submerge it into the tournament tree. To make *minimum* straightforward, we recall the position of the overall minimum of these three components.

In Figure 2, we describe in pseudo-code how a bucket is submerged into the tournament tree at one go. In *insert*, the submersion is done incrementally. The process starts from a leaf corresponding to the given bucket and proceeds in a bottom-up manner to the root by following the implicit parent pointers. We call the path of nodes visited the *updating path*. First, the index at the leaf is set to point to the minimum of the bucket. Then, every branch node on the updating path inherits the position of the smaller of the two elements pointed to via its two children. At the end

**procedure:** *submersion*

**input:** *bucket-start*: index of the beginning of the bucket to be submerged  
*bucket-min*: index of the minimum alive element within this bucket

**data:**  $N$ : number of elements;  $\bar{S}$ : workspace target rounded to a power of 2  
 $A[0..N-1]$ : read-only array of elements  
 $T[0..2\bar{S}-1]$ : array of indices from  $\{\mathbf{none}, 0, 1, \dots, N-1\}$   
 $header[0.. \lg \bar{S}]$ : array of offsets,  $header[h] = 2^h - 1$  for  $h \in \{0, 1, \dots, \lg \bar{S}\}$

$current \leftarrow bucket\text{-}start / \lceil N/\bar{S} \rceil$   
 $leaf \leftarrow header[\lg \bar{S}] + current$   
 $T[leaf] \leftarrow bucket\text{-}min$

**for**  $\ell \in \{\lg \bar{S}, \lg \bar{S} - 1, \dots, 1\}$ :

$parent \leftarrow \lfloor current/2 \rfloor$   
 $this \leftarrow header[\ell] + current$   
 $sibling \leftarrow \mathbf{if} \text{ } current \bmod 2 = 0: this + 1 \text{ else } this - 1$   
 $above \leftarrow header[\ell - 1] + parent$   
**if**  $T[sibling] = \mathbf{none}$ :  
   $T[above] \leftarrow T[this]$   
**else if**  $T[this] = \mathbf{none}$ :  
   $T[above] \leftarrow T[sibling]$   
**else if**  $A[T[this]] < A[T[sibling]]$ :  
   $T[above] \leftarrow T[this]$   
**else**:  
   $T[above] \leftarrow T[sibling]$   
 $current \leftarrow parent$

FIG. 2. *Submersion when done at one go.*

the root stores the index of the overall minimum among all alive elements. Since, at some point, some buckets may have no alive elements, we use an unspecified constant **none** to indicate that the covered range of the node in question has no alive elements.

The pseudo-code of *insert* is given in Figure 3. At first, the next element from the input array becomes part of the insertion buffer. If the new element is smaller than the buffer minimum and/or the overall minimum, the positions of these minima are updated. Once the insertion buffer becomes full, the submersion buffer must have been already submerged into the tournament tree. At this point, we treat the insertion buffer as the new submersion buffer and start a new incremental submersion process that recomputes the indices at the nodes on the updating path bottom-up, one by one. As long as the submersion is not finished, each *insert* carries out a constant amount of the submersion work. Since the work needed to update this path is  $O(\lg S)$ , which is  $O(N/S)$  when  $S \leq N/\lg N$ , the process terminates before the insertion buffer becomes again full. Clearly, *insert* takes  $O(1)$  worst-case time.

In *extract*, there are three cases depending on whether the bucket that contains the element to be extracted is one of the buffers or is covered by a node of the tournament tree. However, these cases are quite similar. For a pseudo-code description, see Figure 4. To begin with, the latest output is set up to date. The bucket index of the given element can be determined by simple calculations. Because of the monotonicity assumption the smallest alive element is to be extracted, so the bucket must be scanned to find its new minimum. If the current bucket is the insertion buffer, it is just enough to update the position of its minimum. If the current bucket is the

**procedure:** *insert*  
**input:**  $i$ : index of an element to be inserted into  $T$   
**data:**  $N$ : number of elements;  $\bar{S}$ : workspace target rounded to a power of 2  
 $A[0..N-1]$ : read-only array of elements  
 $T[0..2\bar{S}-1]$ : array of indices from  $\{\mathbf{none}, 0, 1, \dots, N-1\}$   
 $\text{insertion-buffer-start}$ : index of the beginning of the insertion buffer  
 $\text{insertion-buffer-size}$ : number of elements in the insertion buffer  
 $\text{insertion-buffer-min}$ : index of the minimum of the insertion buffer  
**assert:**  $\text{insertion-buffer-start} + \text{insertion-buffer-size} = i$   
 $\text{insertion-buffer-size} \leftarrow \text{insertion-buffer-size} + 1$   
**if**  $\text{insertion-buffer-min} = \mathbf{none}$  **or**  $A[i] < A[\text{insertion-buffer-min}]$ :  
     $\text{insertion-buffer-min} \leftarrow i$   
**if**  $\text{overall-min} = \mathbf{none}$  **or**  $A[i] < A[\text{overall-min}]$ :  
     $\text{overall-min} \leftarrow i$   
**if**  $\text{insertion-buffer-size} = \lceil N/\bar{S} \rceil$ :  
     $\text{submersion-buffer-start} \leftarrow \text{insertion-buffer-start}$   
     $\text{submersion-buffer-size} \leftarrow \text{insertion-buffer-size}$   
     $\text{submersion-buffer-min} \leftarrow \text{insertion-buffer-min}$   
     $\text{insertion-buffer-start} \leftarrow i + 1$   
     $\text{insertion-buffer-size} \leftarrow 0$   
     $\text{insertion-buffer-min} \leftarrow \mathbf{none}$   
execute  $O(1)$  steps of the submersion process, if one is in progress

FIG. 3. *Inserting an element into an adjustable tournament tree.*

submersion buffer, the submersion process is completed by recomputing the indices at the nodes on the updating path covering the submersion buffer. Hereafter the submersion buffer ceases to exist. If the current bucket is covered by the tournament tree, it is necessary to recompute the indices at nodes on the updating path covering the current bucket. At the end, the position of the overall minimum is to be updated. It is the scanning of a bucket that makes this operation expensive: The worst-case running time is  $O(N/S + \lg S)$ , which is  $O(N/S)$  when  $S \leq N/\lg N$ .

**2.3. Adjustable Navigation Piles.** In brief, an *adjustable navigation pile* is a compact representation of an adjustable tournament tree. The main differences are as follows (compare Figure 1 and Figure 5):

1. A bit vector of size  $2\bar{S} - 1$  is used to indicate whether or not the buckets covered by each node contain any alive elements. As for the data in the tournament tree, in this bit vector the bits of the nodes are stored in breadth-first order. This bit vector can be used to emulate the constant **none**.
2. Only branch nodes, i.e. nodes whose heights are larger than 0, store some additional information about the position of the smallest alive element in the covered range of each of these nodes. In the complete binary tree built above the buckets, the number of branch nodes is  $\bar{S} - 1$ .
3. The navigation information is stored in a bit vector of size  $4\bar{S}$  (an explanation will follow shortly). To save space, at the bottom of the tree, the position of the smallest alive element is only specified approximately. Here the details of our construction differ from those used in the navigation piles of [28] and their precursors [37], although the techniques used are similar.

**procedure:** *extract*  
**input:**  $j$ : index of an element to be extracted from  $T$   
**data:**  $N$ : number of elements;  $\bar{S}$ : workspace target rounded to a power of 2  
 $A[0..N-1]$ : read-only array of elements  
 $T[0..2\bar{S}-1]$ : array of indices from  $\{\mathbf{none}, 0, 1, \dots, N-1\}$   
*insertion-buffer-start*: index of the beginning of the insertion buffer  
*insertion-buffer-min*: index of the minimum of the insertion buffer  
*submersion-buffer-start*: index of the beginning of the submersion buffer  
*submersion-buffer-min*: index of the minimum of the submersion buffer

$\text{latest-output} \leftarrow A[j]$   
 $\text{bucket-start} \leftarrow \lceil N/\bar{S} \rceil \cdot \lfloor j / \lceil N/\bar{S} \rceil \rfloor$   
 $\text{bucket-min} \leftarrow \mathbf{none}$

**for**  $i \in \{\text{bucket-start}, \text{bucket-start} + 1, \dots, \min\{\text{bucket-start} + \lceil N/\bar{S} \rceil - 1, N-1\}\}$ :  
    **if**  $\text{latest-output} < A[i]$  **and** ( $\text{bucket-min} = \mathbf{none}$  **or**  $A[i] < A[\text{bucket-min}]$ ):  
         $\text{bucket-min} \leftarrow i$

**if**  $\text{bucket-start} = \text{insertion-buffer-start}$ :  
     $\text{insertion-buffer-min} \leftarrow \text{bucket-min}$

**else if**  $\text{bucket-start} = \text{submersion-buffer-start}$ :  
     $\text{submersion-buffer-min} \leftarrow \text{bucket-min}$   
     $\text{submersion}(\text{submersion-buffer-start}, \text{submersion-buffer-min})$   
     $\text{submersion-buffer-min} \leftarrow \mathbf{none}$

**else:**  
     $\text{submersion}(\text{bucket-start}, \text{bucket-min})$

$\text{overall-min} \leftarrow \mathbf{none}$

**for**  $k \in \{T[0], \text{submersion-buffer-min}, \text{insertion-buffer-min}\}$ :  
    **if**  $k \neq \mathbf{none}$  **and** ( $\text{overall-min} = \mathbf{none}$  **or**  $A[k] < A[\text{overall-min}]$ ):  
         $\text{overall-min} \leftarrow k$

FIG. 4. *Extracting an element from an adjustable tournament tree.*

A branch node of height  $h \in \{1, 2, \dots, \lg \bar{S}\}$  covers  $2^h$  buckets. As in a navigation pile [28], due to scarcity of bits, the bucket index is *relative* within the covered range. We use  $h$  bits to specify in which bucket the smallest alive element is. A significant new ingredient, borrowed from [37], is the concept of a *quantile*. For a branch node of height  $h$ , every covered bucket is divided into  $2^h$  quantiles, and we store additional  $h$  bits to specify in which quantile the smallest alive element is. We call this quantile the *active quantile* of the node. A quantile contains  $\lceil N/(\bar{S} \cdot 2^h) \rceil$  elements, except that the last quantile can possibly be smaller. Thus, we use  $2h$  bits per node; but if  $2h \geq \lceil \lg N \rceil$ , we only use  $\lceil \lg N \rceil$  bits (since this is enough to specify the exact position of the smallest alive element). To sum up, since there are  $\bar{S}/2^h$  nodes of height  $h$  and since at each node we store  $\min\{2h, \lceil \lg N \rceil\}$  bits, the total number of navigation bits is bounded by

$$\sum_{h=1}^{\lg \bar{S}} \frac{\bar{S} \cdot \min\{2h, \lceil \lg N \rceil\}}{2^h} < 4\bar{S}.$$

The navigation bits are stored in a bit vector in breadth-first order. As before, we maintain a *header* giving the position of the first bit at each level. The space needed by the header is  $O(\lg^2 \bar{S})$  bits. Inside each level, the navigation information



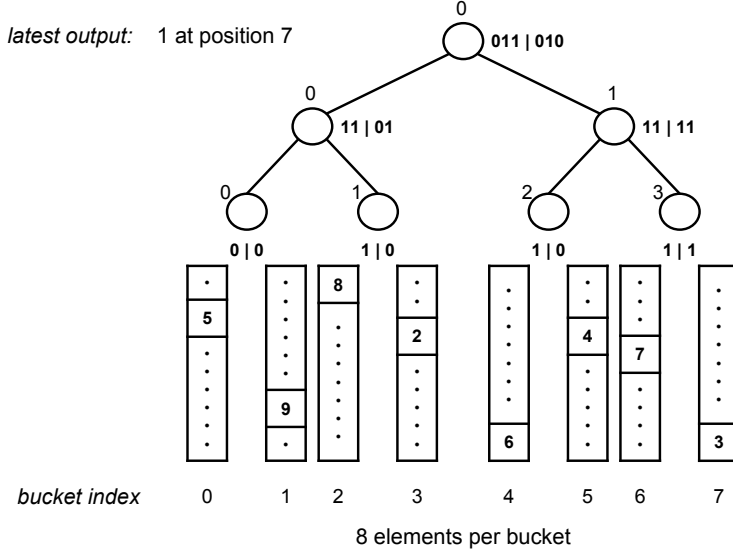


FIG. 5. An adjustable navigation pile corresponding to the adjustable tournament tree in Figure 1. In this snapshot all buckets contain one or more alive elements so the bit vector—that is not shown—indicating their existence contains just 1 bits.

is stored compactly side by side, and the nodes are numbered at each level starting from 0. Since the length of the navigation bits is fixed for all nodes at the same level, using the height and the index of a node, it is easy to calculate the positions where the navigation bits of that node are stored.

Let us now consider how to access the desired quantile for a branch node. The procedure is summarized in pseudo-code form in Figure 6. Let the branch node be at index  $v$  within its own level, and assume that its height is  $h$ . The first element of the covered range of this branch node is in position  $v \cdot 2^h \cdot \lceil N/\bar{S} \rceil$ . The first  $h$  bits of the navigation information give the desired bucket inside the covered range; let the relative bucket index be  $b$ , so we have to go  $b \cdot \lceil N/\bar{S} \rceil$  positions forward. The second  $h$  bits of the navigation information give the desired quantile inside that bucket; let this quantile be  $q$ , so we have to proceed another  $q \cdot \lceil N/(\bar{S} \cdot 2^h) \rceil$  positions forward before we reach the beginning of the desired quantile. Obviously, these calculations can be carried out in constant time.

The priority-queue operations are implemented in a similar way as for an adjustable tournament tree. To facilitate constant-time *minimum*, we keep in memory the index of the overall minimum (since the root of an adjustable navigation pile does not necessarily specify a single element). In *insert* and *extract*, the subtle difference is dealing with quantiles. When updating the navigation information for a branch node, at the bottom of an adjustable navigation pile, we do not have direct access to the minimum among the alive elements covered. Instead, we have to scan the elements in the quantiles specified for the sibling nodes of the nodes along the updating path. Later on, a quantile is said to be *active* if it contains the minimum among the alive elements covered by a node. After updating the navigation bits of a node  $v$ , we locate its parent node  $u$  and its sibling node  $w$ . The navigation bits of  $w$  are used to locate its active quantile. This quantile is scanned, and the minimum of the alive elements is found and compared with the minimum of the alive elements covered by  $v$ . From

**procedure:** *calculate-quantile*  
**input:**  $v$ : index of a branch node at its own level;  $h$ : height of that node  
**data:**  $N$ : number of elements;  $\bar{S}$ : workspace target rounded to a power of 2  
 $B[0..4\bar{S}-1]$ : array of navigation bits  
 $header[1.. \lg \bar{S}]$ : array of offsets  
 $\lambda \leftarrow \min\{2h, \lceil \lg N \rceil\}$   
 $info-start \leftarrow header[h] + v \cdot \lambda$   
**if**  $\lambda = \lceil \lg N \rceil$ :  
     $quantile-start \leftarrow B[info-start..info-start + \lceil \lg N \rceil - 1]$   
     $quantile-size \leftarrow 1$   
    **return** ( $quantile-start, quantile-size$ )  
 $b \leftarrow B[info-start..info-start + h - 1]$   
 $q \leftarrow B[info-start + h..info-start + 2 \cdot h - 1]$   
 $covered-range-start \leftarrow v \cdot 2^h \cdot \lceil N/\bar{S} \rceil$   
 $bucket-start \leftarrow covered-range-start + b \cdot \lceil N/\bar{S} \rceil$   
 $quantile-start \leftarrow bucket-start + q \cdot \lceil N/(\bar{S} \cdot 2^h) \rceil$   
 $quantile-past \leftarrow \min\{quantile-start + \lceil N/(\bar{S} \cdot 2^h) \rceil, bucket-start + \lceil N/\bar{S} \rceil\}$   
 $quantile-size \leftarrow quantile-past - quantile-start$   
**return** ( $quantile-start, quantile-size$ )

FIG. 6. Calculating the beginning and size of the active quantile of a node.

the bucket index and the position of the smaller of the two elements, the navigation bits of  $u$  are then calculated and accordingly updated. If the active quantile of  $u$  has only one element, the position of this single element can be stored as such.

The key point is that for a node of height  $h$  the size of the active quantile is at most  $\lceil N/(\bar{S} \cdot 2^h) \rceil$ , so the total work done in the scans of the quantiles of the siblings along the updating path is proportional to  $\sum_{h=1}^{\lg \bar{S}} \lceil N/(\bar{S} \cdot 2^h) \rceil$ , which is  $O(N/S + \lg S)$ . It follows that the asymptotic efficiency of the priority-queue operations is the same as for an adjustable tournament tree.

**2.4. Getting Rid of the Assumptions.** So far we have consciously ignored the fact that the sizes of the buckets depend on  $N$ , and that we might not know this value beforehand. The standard way of handling this is to rely on global rebuilding [36, Chapter V]. We use an estimate  $N_0$  and initially set  $N_0 = 8$ . We build two data structures, one for  $N_0$  and another for  $2N_0$ . The first structure is used to perform the priority-queue operations, but insertions and extractions are mirrored in the second structure (if the extracted element exists there). When the structure for  $N_0$  becomes too small, we dismiss the smaller structure in use, double  $N_0$ , and in accordance start building a new structure of size  $2N_0$ . We should speed up the construction of the new structure by inserting up to two alive elements into it at a time, instead of only one. This guarantees that the new structure will be ready for use before the first one is dismissed. Even though global rebuilding makes the construction more complicated, the time and space bounds remain asymptotically the same.

Since we have random-access capability to the read-only input, it is not necessary that elements are inserted by visiting the input sequentially, but insertions should still be monotonic. If this is the case, in connection with each *insert*, we have to fix the information related to the current bucket as in *extract*. That is, we have to find the smallest alive element of the bucket and, if the inserted element is smaller than the

```

procedure: priority-queue-sort
input:  $A[0..N-1]$ : read-only array of  $N$  elements;  $S$ : workspace target
 $P \leftarrow \text{navigation-pile}(A, S)$ 
for  $i \in \{0, 1, \dots, N-1\}$ :
     $P.\text{insert}(i)$ 
while  $|P| > 0$ :
     $j \leftarrow P.\text{minimum}()$ 
     $P.\text{extract}(j)$ 
     $\text{print}(A[j])$ 

```

FIG. 7. Priority-queue sort; the position of an element is specified by its index.

current minimum at that bucket, update the navigation information on the updating path covering that bucket. The worst-case cost of *insert* then becomes the same as that of *extract*, i.e.  $O(N/S + \lg S)$ .

In some applications, insertions and extractions may not be monotonic. To handle this situation, we have to allocate one bit per array entry ( $N$  bits in total), indicating whether the corresponding element is alive or not.

### 3. Sorting.

**3.1. The Pagter-Rauhe Algorithm.** Let us turn our attention to sorting. Given  $N$  elements in a read-only array, the task is to print the elements in non-decreasing order. Assume that the asymptotic workspace target is  $S$ . In the basic setting, Pagter and Rauhe [37] proved that the running time of their sorting algorithm is  $O(N^2/S + N \lg^2 S)$  using  $O(S)$  bits of workspace. Lagging behind the optimal bound for the space-time product by a logarithmic factor when  $S = \omega(N/\lg^2 N)$ , they suggested using their memory-adjustable data structure in Frederickson's adjustable binary heap [21] to handle subproblems of size  $(N \lg N)/S$  using  $O(\lg N)$  bits for each. In accordance, by combining the two data structures, they achieve an optimal  $O(N^2/S)$  running time for sorting when  $\lg N \leq S \leq N/\lg N$ . In our treatment we avoid the complication of plugging two data structures together.

**3.2. Priority-Queue Sort.** To sort the elements, we create an empty adjustable navigation pile, insert the elements into this pile by scanning the read-only array from beginning to end, and then repeatedly extract the minimum of the remaining elements from the pile. The pseudo-code in Figure 7 implements this algorithm.

**3.3. Analysis.** From the bounds derived for the priority queue, the asymptotic performance can be directly deduced: The worst-case running time is  $O(N^2/S + N \lg S)$  and the size of workspace is  $O(S)$  bits, where  $S \geq \lg N$ . It is also easy to count the number of element comparisons performed during the execution of the algorithm. When inserting the  $N$  elements into the data structure,  $O(N)$  element comparisons are performed. We can assume that after these insertions, the buffers are submerged into the main structure. In each *extract* we have to find the minimum of a single bucket which requires at most  $N/\bar{S}$  element comparisons. In addition, we have to update a single path in the complete binary tree. At each level, the minimum below the current node is already known and we have to scan the quantiles of the sibling nodes. During the path update, we have to perform at most  $N/\bar{S} + \lg \bar{S}$  element comparisons. Hence, the total number of element comparisons performed is bounded by  $2N^2/\bar{S} + N \lg \bar{S} + O(N)$ , which is  $2N^2/S + N \lg S + O(N)$  since  $S \leq \bar{S} \leq 2S$ .

## 4. Augmenting the Adjustable Navigation Pile.

**4.1. Motivation for Augmentation.** In our algorithm for computing the convex hull of a set of planar points (see Section 5), insertions are neither monotonic nor sequential and extractions are not monotonic. Nevertheless, we want to keep the algorithm memory adjustable and use  $O(S)$  bits of extra space for  $S < N$ . To limit the size of working space, our solution is to work with a subset of the input constituting at most  $S$  elements at a time. In more details, the input is processed in a number of rounds, where in each round we shall have two filter elements and only elements whose values are between these filters are to be inserted or extracted from the adjustable navigation pile. We refer to these  $S$  elements as the *candidates*. Note that the candidates need not be contiguous in the read-only input array. In every round, we use a vector of  $S$  bits, one bit per candidate, to indicate whether each of these candidates is still alive or has been deleted. Subsequently, we need to map the indices of the input array to indices in the range  $[0 \dots S - 1]$ . To be able to do that, the adjustable navigation pile needs to be augmented with additional information. We shall refer to the adjustable navigation pile explained in Section 2 as the *unaugmented navigation pile* to distinguish it from the augmented version to be described in this section. Next, we introduce the additional structures used in the augmentation. Then, we explain how to update and utilize these structures in the priority-queue operations. A schematic view illustrating the components of an augmented navigation pile is given in Figure 8.

**4.2. Additional Structures.** For the sake of simplicity, assume that the required workspace target  $S$  is a power of 2. We augment the adjustable navigation pile with the following data structures:

- A bit vector *alive* of size  $S$  is used to denote whether each candidate is currently alive or not. The order of the candidates in this vector is identical to their order in the read-only array. The *alive* vector is dynamically updated by *insert* and *extract* operations.
- A bit vector *start* of size  $S$ —corresponding to the same elements, in the same order, as *alive*—is used to denote whether a candidate is the first, among other candidates, of an active quantile or not. We refer to the candidate corresponding to the first entry for a candidate from quantile  $q$  as *first-candidate*( $q$ ). (Note that a candidate may simultaneously be the first candidate of more than one active quantile.) The *start* vector is also dynamic, as every *insert* and *extract* may change it. Every bucket will possibly map to a part of *alive* and *start*, which obviously has at most  $\lceil N/S \rceil$  entries. We refer to the part of *start* corresponding to bucket  $u$  as *start.part*( $u$ ).
- A static bit vector *count* is used to store the number of candidates contained in each bucket. We encode these counts in unary, using a 0 bit to mark the border between every two consecutive buckets. Since we are dealing with at most  $S$  candidates, the vector contains at most  $S$  ones; and since we have exactly  $S$  buckets, it contains  $S - 1$  zeros. The *count* vector should efficiently support *rank* and *select* queries. The bit vector and the accompanying rank-select structures thus consume  $\Theta(S)$  bits. The objective is to efficiently locate the first entry of any bucket in *alive* and *start* using the index of the bucket. Assume the first bucket has index 0. Let  $u > 0$  be the index of the bucket whose first entry in *alive* and *start* is to be located. Then,  $t = \text{count.select0}(u)$  is the index of the last 0 bit preceding bucket  $u$  in *count*. It follows that  $t - u + 1$  is the number of candidates lying in the buckets  $0, 1, \dots, u - 1$ , which precede the first entry of bucket  $u$  in *alive* and *start*. The size (number of bits) of

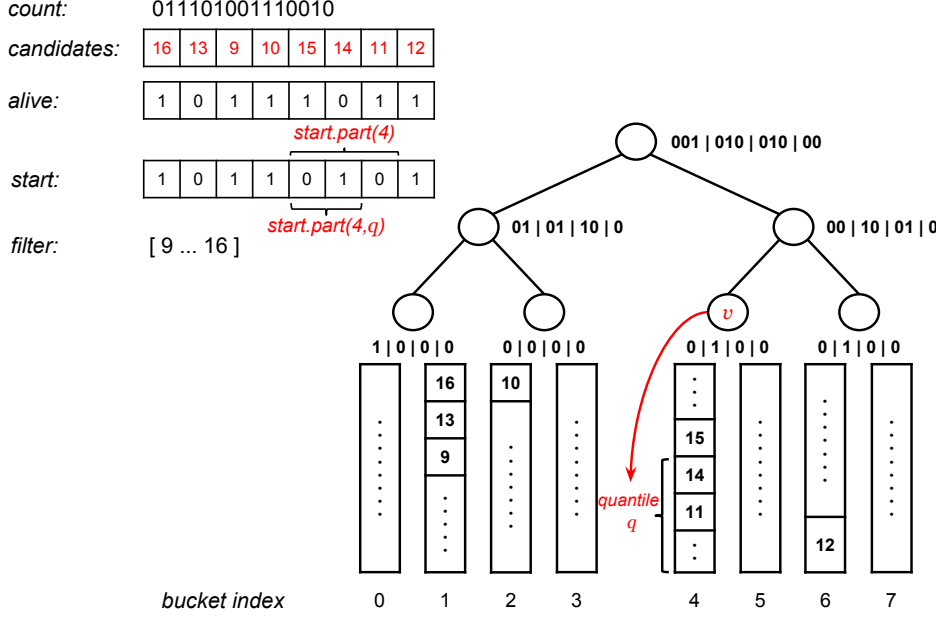


FIG. 8. A snapshot of an augmented navigation pile that has  $N = 64$  and  $S = 8$ . Only elements within the range of the filters are shown. The snapshot is taken considering the given alive vector. For each node, the navigation bits are displayed in left-to-right order as follows: relative bucket index, quantile index,  $start.part$  index, and  $start.before$ . As an example, we can see that node  $v$  refers to the candidate 11 as its minimum alive element. Hence, its relative bucket index is 0, which refers to bucket 4. Bucket 4 is partitioned into two quantiles in the view of node  $v$ , where the candidate 11 lies in the second quantile  $q$ . So, the quantile index for node  $v$  is 1.  $start.part(4)$  in  $start$  is dedicated to bucket 4. At node  $v$ ,  $start.part(4)$  is partitioned into two parts. Since the candidate 14 is  $first-candidate(q)$ , and as this candidate is in the first part of  $start.part(4)$ , the  $start.part$  index of node  $v$  is set to 0. The part that contains  $first-candidate(q)$  is referred to as  $start.part(4, q)$ .

- $start.part(u)$  is calculated as  $count.select0(u+1) - count.select0(u) - 1$ .
- For every node of height  $h$ , associated with bucket  $u$  and quantile  $q$ ,  $start.part(u)$  is further divided into  $2^h$  subparts (or less, if its size is less than  $2^h$ ). Another  $h$  bits will be stored in every node of height  $h$  to indicate in which subpart  $first-candidate(q)$  lies. We refer to this subpart as  $start.part(u, q)$ . Since the size of  $start.part(u)$  is at most  $\lceil N/S \rceil$ , the size of  $start.part(u, q)$  is bounded from above by  $\lceil N/(S \cdot 2^h) \rceil$ , which is the size of a quantile.
- For every branch node of height  $h$ , associated with bucket  $u$  and quantile  $q$ , an additional  $\lceil \lg h \rceil$  bits will be used. These bits encode the number of first candidates of other active quantiles that exist in the same subpart  $start.part(u, q)$  before  $first-candidate(q)$ , i.e. the number of ones in  $start.part(u, q)$  preceding the one representing  $first-candidate(q)$ . Let us refer to the value of this number as  $start.before(u, q)$ . The reason we need only  $\lceil \lg h \rceil$  bits to store this information for each node of height  $h$  is that only quantiles tied to nodes (whose heights are less than  $h$ ) on one and only one path from a leaf node to the node associate with  $q$  can have their first candidates before  $first-candidate(q)$  in  $start.part(u, q)$ .

It directly follows, using simple calculations, that the space complexity of the augmented navigation pile is still  $\Theta(S)$  bits.

To keep the time complexity for *insert* and *extract* in  $O(N/S + \lg S)$ , we need to know, in an efficient way, if a given candidate that belongs to an active quantile is alive or not. Starting with the index of a candidate in the read-only array, we want to find, without altering the time bounds, the index of the corresponding bit in *alive*. Given a node  $v$ , associated with bucket  $u$  and quantile  $q$ , the index of *first-candidate*( $q$ ) in *alive* is to be located. Using the *count* vector and the navigation bits of node  $v$ , we can easily locate *start.part*( $u, q$ ), where the entry we are searching for lies. We also get the value  $r = \text{start.before}(u, q)$  from the navigation bits of node  $v$ . As previously stated, the size of *start.part*( $u, q$ ) is at most the size of quantile  $q$ . While visiting node  $v$ , we shall be scanning quantile  $q$  anyhow. A scan of *start.part*( $u, q$ ) would then not alter the worst-case asymptotic time complexity. We scan *start.part*( $u, q$ ) to find the  $r$ th one bit, the index of the entry at which we find this bit is the index of *first-candidate*( $q$ ) in *alive*.

We always access the elements of a quantile sequentially. To locate the corresponding elements in *alive*, we start at the first entry of the quantile in *alive* as explained above. While scanning the quantile, we repeatedly check the elements one after another. If the next element is a candidate (lying in the range of the filters), we increase the current index to the next entry in *alive* to correspond to this candidate. The same procedure can be applied on buckets. We get the first entry of the bucket in *alive* using the *count* vector, and then move sequentially on the bucket and on *alive*, increasing the *alive* index whenever we encounter the next candidate in the bucket.

**4.3. Operations.** We next explain how *insert* and *extract* can be performed in our augmented navigation pile in  $O(N/S + \lg S)$  worst-case time per operation.

We first find the bucket in which the element to be inserted or extracted lies; this can be done with simple calculations in constant time once we have the array index of the element. Let the index of this bucket be  $u$ . Using the *count* vector, we get the first entry of this bucket in *alive* as explained earlier, and move sequentially on bucket  $u$  and *alive*. We can then get the indices of the candidates lying in this bucket within *alive*, and consequently know whether each of these candidates is currently alive or not. After knowing the index of the element to be inserted or extracted in *alive*, we should set the corresponding bit to 1 or 0 respectively. Also, while scanning the bucket, we would know if this element is the minimum in the bucket or not. If the minimum alive element in bucket  $u$  has changed due to the current operation, the following updates need to be done.

As in an unaugmented navigation pile, the information in the nodes along the updating path is to be fixed bottom up. The update will work as in the unaugmented navigation pile, where we scan the quantiles associated with the nodes lying on or hanging from the updating path. However, here we also want to know whether each of the candidates in these quantiles is alive or not. Before accessing a quantile, we get its first entry in *alive*; this can be done as previously described. Then, we simultaneously scan both the quantile and the corresponding subpart in *alive*.

Next, we show how to update the *start* vector. Suppose that we are to handle a node  $v$ , associated with quantile  $q$ , on the updating path, knowing that its child has just been handled. If the index of *first-candidate*( $q$ ) is different from the index of the first candidate of each of the two quantiles associated with the two children of node  $v$ , we reset the bit for *first-candidate*( $q$ ) in *start* to 0. Note that we do not reset that bit to 0 if it is the first entry of the quantile of a child of node  $v$  in *start*. Alternatively, this bit may be temporarily reset to 0 in the previous step and then again set to 1 within the upcoming step. Assume that the child of node  $v$  that is associated with

the quantile that has the smaller element of the two children of  $v$  associated with quantile  $q'$ . Now the quantile associated with node  $v$  should be either the first half or the second half of quantile  $q'$ . If it is the first half, then the corresponding entry in  $start$  must have been already set to 1 before. Else, we move sequentially on quantile  $q'$  and  $start$  till we reach the first candidate in the second half of quantile  $q'$ , and set its corresponding entry in  $start$  to 1. The above procedure is repeated for every node on the updating path.

For the nodes along the updating path, we show next how the additional bits in our augmented navigation pile will be updated. Consider a node  $v$  that is associated with bucket  $u$  and quantile  $q$  after the update. While handling node  $v$ , as explained earlier, we are able to know the entry in  $start$  for  $first-candidate(q)$  as well as the size of  $start.part(u, q)$ . Knowing these values, it is easy to update the bits indicating  $start.part(u, q)$  in node  $v$ . Also, after getting these bits, we loop on  $start.part(u, q)$  to count the number of 1 bits in this subpart preceding the entry for  $first-candidate(q)$  in  $start$ , and store this value for node  $v$  in  $start.before(u, q)$ .

It is obvious that the update will be performed on at most  $\lg S$  nodes on the updating path. Also, looping on the quantiles and the corresponding parts of  $start$  for the nodes on the updating path would sum up to  $O(\sum_{i=1}^{\lg S} [N/(S \cdot 2^i)])$ . So the time complexity for the update is in  $O(N/S + \lg S)$ , as claimed.

The following lemma summarizes the functionality of our data structure.

**LEMMA 1.** *Assume that we are given a read-only array of  $N$  elements, and two filters that enclose at most  $S$  of these elements between their values. Using  $\Theta(S)$  bits of workspace, after spending  $O(N)$  worst-case time on building the augmented structure, insert and extract can be applied to any element of the array whose value is between the filters in  $O(N/S + \lg S)$  worst-case time per operation.*

## 5. Convex Hulls.

**5.1. The Chan-Chen Algorithm.** Consider now the problem of computing the convex hull of a set of  $N$  points in the plane given in a read-only array  $P$ . Without loss of generality, we can assume that the points are unique such that no two points have the same  $x$ -coordinate or  $y$ -coordinate. The task is to print the points on the convex hull in the clockwise order of their appearance on the hull, starting from some arbitrary point. As is standard, it is enough to show how to compute the upper hull of the point set; the lower hull can be computed in a symmetric manner. We assume the availability of the standard geometric primitive that tells whether or not there is a right turn on point  $P[j]$  when going from point  $P[i]$  to point  $P[k]$  via  $P[j]$ ; we denote this predicate as  $right-turn(P[i], P[j], P[k])$ .

A high-level description of the algorithm by Chan and Chen [9] is given in Figure 9. Our algorithm is similar but the details are different. When the working space of  $\Theta(S)$  bits is available, they set the space parameter  $s$  to  $S/\lg N$  and use  $\Theta(s)$  indices to recall the points being processed. The algorithm performs  $\lceil N/s \rceil$  rounds and handles the  $s$  points with the next smallest  $x$ -coordinates in each round; these points form a vertical slab  $\sigma$ . In each round, the algorithm starts with a known hull vertex  $P[i_0]$  and computes the part of the upper hull for the points of  $\sigma$  starting from point  $P[i_0]$  and ending at the left endpoint of the hull edge crossing the right wall of  $\sigma$ .

When finding the  $s$  points with the next smallest  $x$ -coordinates among the remaining points, the algorithm uses space for  $2s$  indices and maintains in the first half the  $s$  indices of the points with the smallest  $x$ -coordinates among the points examined so far. Each time, the second half is refilled with another  $s$  indices from the unexamined portion, the median of the  $x$ -coordinates of the  $2s$  recorded points is

```

procedure: compute-convex-hull
input:  $P[0..N-1]$ : read-only array of  $N$  points;  $s$ : workspace target
 $i_0 \leftarrow \text{none}$ 
for  $i \in \{0, 1, \dots, N-1\}$ :
    if  $i_0 = \text{none}$  or  $x\text{-coordinate}(P[i]) < x\text{-coordinate}(P[i_0])$ :
         $i_0 \leftarrow i$ 
while  $i_0 \neq \text{none}$ :
    let  $\sigma$  be the vertical slab containing  $P[i_0]$  and the next  $s-1$  points (if possible)
        on the right of and closest to the wall determined by  $P[i_0]$ 
    let  $\{i_0, i_1, \dots, i_{s-1}\}$  be the indices of the  $s$  points in  $\sigma$ , sorted by  $x$ -coordinate
    use Graham's scan to compute the upper hull of these points
    let  $\{h_0, h_1, h_2, \dots, h_{s'}\}$  be the indices of the points on this hull //  $h_0 = i_0$ 
     $j' \leftarrow \text{none}$  // right endpoint of the hull edge crossing the right wall of  $\sigma$ 
    for each point  $P[j]$  to the right of the wall determined by  $P[i_{s-1}]$ :
        if  $j' \neq \text{none}$  and  $\text{right-turn}(P[h_{s'}], P[j'], P[j])$ :
            continue
        while  $s' > 0$  and not  $\text{right-turn}(P[h_{s'-1}], P[h_{s'}], P[j])$ :
             $s' \leftarrow s' - 1$ 
         $j' \leftarrow j$ 
    for  $k \in \{h_0, h_1, \dots, h_{s'}\}$ :
         $\text{print}(P[k])$ 
     $i_0 \leftarrow j'$ 

```

FIG. 9. High-level description of the Chan-Chen algorithm.

found, and these points are partitioned around this median. This is repeated until all the points are examined, leaving the  $s$  points within the slab  $\sigma$ . The upper hull of these points can then be constructed using any of the known  $O(s \lg s)$  convex-hull algorithms [38, Chapter 3]; a natural choice is to use Graham's scan since the rest of the algorithm follows the same elimination strategy. The Chan-Chen algorithm eliminates the points that are not on the convex hull by traversing the tentative hull chain in reverse order, the points with the larger  $x$ -coordinates first. This procedure is done through finding the hull edge crossing the right wall of  $\sigma$  by performing a pass over the remaining points (those to the right of  $\sigma$ ). Suppose a point  $P[j]$  from the remaining points is currently being inspected, by imitating Graham's scan, the point  $P[j]$  is tentatively added to the hull if it is above the tentative hull edge found so far crossing the right wall of  $\sigma$ . Also, adding  $P[j]$  to the hull might require removing some points from the chain in Graham-scan fashion. In the description given in Figure 9, we do not add any new points to the tentative hull chain. We just keep  $P[j']$  as the point representing the right endpoint of the hull edge crossing the right wall of  $\sigma$  and we update  $j'$  accordingly. Hence, points are only removed from the hull chain during the elimination process. At the end of the pass, the leftover points on the chain are indeed on the convex hull and are accordingly reported.

Finding the next  $s$  points with the smallest  $x$ -coordinates requires  $O(N)$  time, computing the upper hull for  $s$  points requires  $O(s \lg s)$  time, and pruning the tentative hull chain from points not on the upper hull requires  $O(N)$  time. Since there are  $O(N/s)$  rounds, the algorithm runs in  $O(N/s \cdot (N + s \lg s))$  worst-case time, which is  $O((N^2/S) \cdot \lg N + N \lg S)$ .



**5.2. Our Algorithm.** Our convex-hull algorithm uses three navigation piles: a (max-)augmented navigation pile that we call *max-pile*, and two (min-)unaugmented navigation piles that we call *min-pile*<sub>1</sub> and *min-pile*<sub>2</sub>. Without loss of generality, we assume that the orientation (either max or min) of the navigation piles is with respect to the  $x$ -coordinates of the points. The algorithm works as follows:

1. Insert all the  $N$  points in both *min-pile*<sub>1</sub> and *min-pile*<sub>2</sub>.
  2. Let  $i_0$  be the index of the point with the minimum  $x$ -coordinate, and **none** if the input set is empty.
  3. **while**  $i_0 \neq \text{none}$ :
    - (a) Extract the minimum  $S$  points, one by one, from *min-pile*<sub>1</sub> (or until  $|\text{min-pile}_1| = 0$ ), and keep track of the first and last points. These two values will be used as filters for *max-pile*; let us call them  $f_1$  and  $f_2$ . These  $S$  points will be the candidates considered in this round, determining the slab  $\sigma$ .
    - (b) Reinitialize an empty *max-pile* using the current candidates and filters, without actually inserting the points. Reinitialize the *count* vector by scanning the  $N$  points, bucket by bucket. Then, build the *rank* and *select* structures for the *count* vector.
    - (c) Construct the upper hull for the current  $S$  points. This can be done by using a space-efficient implementation of Graham's scan. For example, the in-place variant described in [6] could be modified to use a bit vector of alive elements, instead of swapping the input elements. In this computation, *max-pile* and *min-pile*<sub>2</sub> can be deployed as follows.
      - i. Extract the two points with the minimum  $x$ -coordinates from *min-pile*<sub>2</sub> and insert both of them into *max-pile*.
      - ii. **repeat**  $S - 2$  times (or until  $|\text{min-pile}_2| = 0$ ):
        - A. Extract the minimum point from *min-pile*<sub>2</sub>; let its index be  $i'$ .
        - B. **while**  $|\text{max-pile}| \geq 2$  **and not** *right-turn*(next to maximum of *max-pile*, maximum of *max-pile*,  $P[i']$ ):  
repeatedly extract the maximum point from *max-pile*.
        - C. Insert the point  $P[i']$  into *max-pile*.
- At this point, the alive points in *max-pile* form a tentative upper hull.
- (d) The goal here is to eliminate the points that are not really on the hull among the points in *max-pile* forming the tentative hull chain. We do so by computing the hull edge crossing the right wall of  $\sigma$ .
    - i. Set  $j'$  to **none**, which represents the right endpoint of the hull edge crossing the right wall of  $\sigma$ .
    - ii. **for** every point with index  $j$  whose  $x$ -coordinate is greater than  $f_2$ :
      - A. If  $j' \neq \text{none}$  **and** *right-turn*(maximum of *max-pile*,  $P[j']$ ,  $P[j]$ ), **continue**.
      - B. **while**  $|\text{max-pile}| \geq 2$  **and not** *right-turn*(next to maximum of *max-pile*, maximum of *max-pile*,  $P[j]$ ):  
repeatedly extract the maximum point from *max-pile*.
      - C. Set  $j'$  to  $j$ .
  - (e) Set  $i_0$  to  $j'$ , then extract and neglect points from *min-pile*<sub>1</sub> and *min-pile*<sub>2</sub> until we reach  $P[i_0]$  as the current minimum for both.
  - (f) Extract all alive points from *max-pile* and report them as points on the convex hull, but in reverse order. This reversal can be done by inserting the points in an empty min-navigation pile, then extracting them.

By inspecting our algorithm and the algorithm of Chan and Chen, at high level, the algorithms are quite similar. The main difference is that we use adjustable navigation piles in order to recall the points and deal with them. It should also be noted that the full power of our techniques (augmenting the structure) is only needed in step (3c). All other parts could have been handled efficiently without augmenting the adjustable navigation pile.

**5.3. Analysis.** We need to prove that our convex-hull algorithm achieves a time complexity of  $O(N^2/S + N \lg S)$  and a space complexity of  $\Theta(S)$  bits. As the space complexity, we deploy three navigation piles that are proved to require  $\Theta(S)$  bits. As for the time complexity, step (1) requires  $O(N)$  time to build two unaugmented navigation piles. Obviously, step (2) can be done in  $O(1)$  time. Now, we are going to analyse step (3). There are  $\lceil N/S \rceil$  rounds in total. The work done in each round can be summed up as follows:

- In step (3a), extracting  $S$  points requires  $O(N + S \lg S)$  time.
- Step (3b) needs  $O(N)$  time to construct the *count* vector and the data structures for answering *rank* and *select* queries.
- Constructing the upper hull of  $S$  points in step (3c) is done in  $O(N + S \lg S)$  time. Note that  $S$  points are extracted from *min-pile*<sub>2</sub>. Each of these points will be inserted and may be extracted later from *max-pile*, but a point can be inserted once and extracted once from *max-pile*, for a total of at most  $S$  insertions and  $S$  extractions.
- The time complexity of step (3d) is  $O(N + S \lg S)$  too. We need  $O(N)$  to loop on the whole input sequence. Also, we may be extracting points from *max-pile*. Given that the number of alive points in *max-pile* is at most  $S$ , then the extractions need at most  $O(N + S \lg S)$  time.
- Step (3e) does not affect the time complexity of the algorithm. Here, we extract points from *min-pile*<sub>1</sub> and *min-pile*<sub>2</sub>. Since these piles initially contain the  $N$  points and no more insertions are done into them, throughout the algorithm all extractions from these piles require  $O(N^2/S + N \lg S)$  time.
- Given that the number of points in *max-pile* is at most  $S$ , step (3f) (including the reversal of the order of the points) will be done in  $O(N + S \lg S)$  time.

As a conclusion, the total cost of step (3e) is  $O(N^2/S + N \lg S)$ . Except for step (3e), the worst-case time complexity of step (3) is  $O(N + S \lg S)$  per round. Multiplying this by the number of rounds  $\lceil N/S \rceil$ , the time complexity of our convex-hull algorithm is  $O(N^2/S + N \lg S)$ , as claimed.

## 6. Concluding Remarks.

**6.1. Summary.** When constructing adjustable navigation piles, four techniques are important: 1) implicit links when indexing different types of objects, 2) bit packing and unpacking, 3) buffering and incremental submersion, and 4) quantile thinning. In addition to their connection to binary heaps [41], we pointed out the strong connection between the navigation piles and tournament trees. Conclusively, our data structure is shown to be a useful ingredient in space-efficient algorithms for problems that employ incremental sorting within their engine. In general, we illustrated some conditions for when a succinct implementation of a priority queue that uses a workspace constituting a sublinear number of bits is possible, so that algorithm designers would be careful when using the structure.

On the negative side, in practice, navigation piles are slow [25] for two reasons: 1) The bit-manipulation machinery is heavy and index calculations devour clock

cycles. 2) The cache behaviour is poor because the memory accesses lack locality. It is expected that the situation is not better for adjustable navigation piles.

Our sorting is a heapsort algorithm [41] that uses an adjustable navigation pile instead of a binary heap. For our convex-hull algorithm we had to augment the data structure with extra information while still maintaining the same asymptotic memory usage. In spite of the optimality of the asymptotic running time of our algorithms, one could criticize the practicality of the model itself, since the memory-access patterns may not always be friendly to contemporary computers while it is not allowed to move the elements around.

**6.2. Related Developments.** The credit for the use of quantile thinning should go to Pagter and Rauhe [37]. However, the way the technique is used in the adjustable navigation pile leads to a simpler and more elegant data structure. Recently, quantile thinning has also been used in a data structure to answer heavy-hitter queries for a set of points on a line [18].

Buffering and incremental submersion is a general data-structural transformation that can be used to speed up *insert* in priority queues [1]. Recently, this technique has also been used in a space-efficient manner in weak heaps [14] and in strengthened lazy heaps [15].

After introducing the adjustable navigation piles in the conference version of this paper, our data structure has also found application in space-efficient graph algorithms [17] and space-efficient plane-sweep geometric algorithms [20].

**6.3. Other Data Structures for Read-Only Data.** In our experience, very few data structures can be made memory adjustable as elegantly as priority queues. A stack is a gratifying companion [4]. As counterweight, a dictionary must maintain a permutation of a set of size  $N$ ; this means that it is difficult to manage with much less than  $N \lceil \lg N \rceil$  bits. However, when the goal is to cope with about  $N$  bits, a bit vector extended with *rank* and *select* facilities (for a survey, see [35]) is a relevant data structure. Two related constructions are the wavelet stack used in [19] and the wavelet tree introduced in [22] (for a survey, see [34]).

**6.4. Open Problems.** The optimality of our algorithm for computing convex hulls follows from the fact that the sorting problem reduces to the problem of computing the convex hull of a planar point set [38, Section 3.2]. However, in the restricted RAM model, for the following variants of the problem, the exact space-time trade-offs are still unknown:

1. Compute the convex hull of a planar set of points given in lexicographic sorted order according to their coordinates.
2. Compute the convex hull of a simple polygon.
3. Compute the extreme points of a planar set of  $N$  *distinct* points, i.e. the points on the convex hull in any order, and express the complexity as the function of  $N$  and  $h$ , where  $h$  denotes the size of the output.

For these problems, the space-time lower bound obtained via the unique-elements problem [40, Section 10.13.7] does not hold any more. For the best known results, we refer to [4, 9] (just be aware that in these papers the space bounds are expressed in words, not in bits).

**Acknowledgements.** We thank Tetsuo Asano for introducing the restricted RAM model to us and taking part in the initial discussions on the topic that led to the post at a conference [3].

## REFERENCES

- [1] S. Alstrup, T. Husfeldt, T. Rauhe, and M. Thorup, Black box for constant-time insertion in priority queues, *ACM Trans. Algorithms* **1**, 1 (2005), 102–106.
- [2] T. Asano, K. Buchin, M. Buchin, M. Korman, W. Mulzer, G. Rote, and A. Schulz, Memory-constrained algorithms for simple polygons, E-print **arXiv:1112.5904**, arXiv.org, Ithaca (2011).
- [3] T. Asano, A. Elmasry, and J. Katajainen, Priority queues and sorting for read-only data, *TAMC 2013, LNCS* **7876**, Springer, Heidelberg (2013), 32–41.
- [4] L. Barba, M. Korman, S. Langerman, K. Sadakane, and R. I. Silveira, Space–time trade-offs for stack-based algorithms, *Algorithmica* **72**, 4 (2015), 1097–1129.
- [5] P. Beame, A general sequential time-space tradeoff for finding unique elements, *SIAM J. Comput.* **20**, 2 (1991), 270–277.
- [6] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. Toussaint, Space-efficient planar convex hull algorithms, *Theoret. Comput. Sci.* **321**, 1 (2004), 25–40.
- [7] S. Carlsson, J. I. Munro, and P. V. Poblete, An implicit binomial queue with constant insertion time, *SWAT 1988, LNCS* **318**, Springer, Heidelberg (1988), 1–13.
- [8] T. M. Chan, Comparison-based time-space lower bounds for selection, *ACM Trans. Algorithms* **6**, 2 (2010), 26:1–26:16.
- [9] T. M. Chan and E. Y. Chen, Multi-pass geometric algorithms, *Discrete Comput. Geom.* **37**, 1 (2007), 79–102.
- [10] T. M. Chan, J. I. Munro, and V. Raman, Selection and sorting in the “restore” model, *SODA 2014*, SIAM, Philadelphia (2014), 995–1004.
- [11] O. Darwish and A. Elmasry, Optimal time-space tradeoff for the 2D convex-hull problem, *ESA 2014, LNCS* **8737**, Springer, Heidelberg (2014), 284–295.
- [12] M. De, S. C. Nandy, and S. Roy, Convex hull and linear programming in read-only setup with limited work-space, E-print **arXiv:1212.5353**, arXiv.org, Ithaca (2012).
- [13] M. De, S. C. Nandy, and S. Roy, Prune-and-search with limited work-space, *J. Comput. System Sci.* **81**, 2 (2015), 398–414.
- [14] S. Edelkamp, A. Elmasry, and J. Katajainen, Weak heaps engineered, *J. Discrete Algorithms* **23** (2013), 83–97.
- [15] S. Edelkamp, A. Elmasry, and J. Katajainen, An in-place priority queue with  $O(1)$  time for push and  $\lg n + O(1)$  comparisons for pop, *CSR 2015, LNCS* **9139**, Springer, Heidelberg (2015), 1–15.
- [16] A. Elmasry, Three sorting algorithms using priority queues, *ISAAC 2003, LNCS* **2906**, Springer, Heidelberg (2003), 209–220.
- [17] A. Elmasry, T. Hagerup, and F. Kammer, Space-efficient basic graph algorithms, *STACS 2015, LIPIcs* **30**, Schloss Dagstuhl—Leibniz Center for Informatics, Dagstuhl (2015), 288–301.
- [18] A. Elmasry, M. He, J. I. Munro, and P. K. Nicholson, Dynamic range majority data structures, *ISAAC 2011, LNCS* **7074**, Springer, Heidelberg (2011), 150–159.
- [19] A. Elmasry, D. D. Juhl, J. Katajainen, and S. R. Satti, Selection from read-only memory with limited workspace, *Theoret. Comput. Sci.* **554** (2014), 64–73.
- [20] A. Elmasry and F. Kammer, Space-efficient plane-sweep algorithms, E-print **arXiv:1507.01767**, arXiv.org, Ithaca (2015).
- [21] G. N. Frederickson, Upper bounds for time-space trade-offs in sorting and selection, *J. Comput. System Sci.* **34**, 1 (1987), 19–26.
- [22] R. Grossi, A. Gupta, and J. S. Vitter, High-order entropy-compressed text indexes, *SODA 2003*, SIAM, Philadelphia (2003), 841–850.
- [23] T. Hagerup, Sorting and searching on the word RAM, *STACS 1998, LNCS* **1373**, Springer, Heidelberg (1998), 366–398.
- [24] G. Jacobson, Space-efficient static trees and graphs, *FOCS 1989*, IEEE Computer Society, Los Alamitos (1989), 549–554.
- [25] C. Jensen and J. Katajainen, An experimental evaluation of navigation piles, CPH STL Report **2006-3**, Dept. Comput. Sci., Univ. Copenhagen, Copenhagen (2006).
- [26] J. Katajainen and T. Pasanen, Sorting multisets stably in minimum space, *Acta Inform.* **31**, 4 (1994), 301–313.
- [27] J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic J. Comput.* **3**, 1 (1996), 27–40.
- [28] J. Katajainen and F. Vitale, Navigation piles with applications to sorting, priority queues, and priority dequeues, *Nordic J. Comput.* **10**, 3 (2003), 238–262.
- [29] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd Edition, Prentice Hall, Englewood Cliffs (1988).

- [30] D. E. Knuth, *Sorting and Searching, The Art of Computer Programming* **3**, 2nd Edition, Addison-Wesley, Reading (1998).
- [31] D. E. Knuth, *Combinatorial Algorithms: Part 1, The Art of Computer Programming* **4A**, Addison-Wesley, Boston (2011).
- [32] J. I. Munro, Tables, *FSTTCS 1996, LNCS* **1180**, Springer, Heidelberg (1996), 37–42.
- [33] J. I. Munro and M. S. Paterson, Selection and sorting with limited storage, *Theoret. Comput. Sci.* **12**, 3 (1980), 315–323.
- [34] G. Navarro, Wavelet trees for all, *CPM 2012, LNCS* **7354**, Springer, Heidelberg (2012), 2–26.
- [35] G. Navarro and E. Proidel, Fast, small, simple rank/select on bitmaps, *SEA 2012, LNCS* **7276**, Springer, Heidelberg (2012), 295–306.
- [36] M. H. Overmars, *The Design of Dynamic Data Structures, LNCS* **156**, Springer, Heidelberg (1983).
- [37] J. Pagter and T. Rauhe, Optimal time-space trade-offs for sorting, *FOCS 1998*, IEEE Computer Society, Los Alamitos (1998), 264–268.
- [38] F. P. Preparata and M. Shamos, *Computational Geometry: An introduction*, Springer, Heidelberg (1985).
- [39] R. Raman, V. Raman, and S. R. Satti, Succinct indexable dictionaries with applications to encoding  $k$ -ary trees, prefix sums and multisets, *ACM Trans. Algorithms* **3**, 4 (2007), 43:1–43:25.
- [40] J. E. Savage, *Models of Computation: Exploring the Power of Computing* (2008). The book is released in electronic form under a CC-BY-NC-ND-3.0-US license and is available at <http://cs.brown.edu/~jes/book/home.html>
- [41] J. W. J. Williams, Algorithm 232: Heapsort, *Commun. ACM* **7**, 6 (1964), 347–348.